CSC469 Assignment 2

Camran Hansen, Andy Chan & Kyoji Goto-Bernier

March 12, 2024

Contents

1 Design

1.A Allocator Choice

When searching the memory allocator literature, multiple aspects of designs were considered. First, as outlined in the *Hoard* paper [1] and the assignment rubric, performance features important to all scalable and memory-efficient allocators were assessed: speed, scalability, false sharing avoidance, and low fragmentation. In addition to these performance features, we also considered more practical implications of allocator design, focusing on how easy the ideas presented in the allocator were to understand, and the estimated amount of code needed to implement them.

One allocator considered was SuperMalloc, which performs favorably compared to Hoard and other allocators such as DLmalloc, JEmalloc, and TBBmalloc on speed, scalability, speed variance, and memory footprint [2]. Additionally, according to analysis done by the authors, it has a small code size of 3,934 lines, compared to 17,056 in Hoard. This made SuperMalloc a promising candidate. However, SuperMalloc does not fulfill the false sharing avoidance criteria outlined in Hoard - citing the assumption that users who care about false sharing will explicitly ask for cache-line alignment, using e.g., `memalign()`. Due to this assumption, we concluded that this allocator design would fail to perform well in benchmarks testing false sharing.

Another allocator considered was MiMalloc, which performs favorably when compared to other allocators on single-threaded speed metrics, and additionally makes considerations to address false sharing, outperforming Hoard and similar allocators on benchmarks such as cscratch [3]. Our other practical considerations are also fulfilled by MiMalloc, which has small code size due to high degrees of code reuse across different block sizes and logical paths, and a relatively simple set of core ideas to avoid issues around concurrency [3]. We anticipated that, in the worst case, ideas proposed in the design such as free list sharding (which will be expanded on in the next section) could be implemented in a simple allocator based on `kheap`, reducing the risk of overcommitting to a complex design and being unable to implement it. Based on this assessment, the core ideas of MiMalloc were chosen as the base structure of the allocator designed for this assignment.

1.B Core Idea: Free List Sharding

The primary idea used in MiMalloc is free list sharding. The allocator uses three page-local sharded free lists to increase locality, avoid contention, and support a highly-tuned allocate and free fast path [3]. Every page, which has a minimum size of 64KiB, has a local free list of equally-sized blocks that can

be allocated to. Whenever a block is freed by the thread that "owns" the page, it is added to the local free list, which does not require any considerations for concurrency. If a non-local thread frees a block, it is atomically moved into the thread-free list for the page. Neither of these free lists (local, thread-free) are merged into the page free-list until there are no available references, resulting in a grouping of maintenance tasks and a reduction of the total number of atomic operations needed for the purposes of synchronization.

1.C Implementation Details

Core Data Structures and Heap Layout In MiMalloc, there are three basic data structures that store memory. Blocks are fixed-size areas of memory that are assigned to objects with a size value that is less than or equal to the block size. Pages, which are similar to a superblock in Hoard, store equal-sized blocks. Three page kinds exist: `small`, sized 64KiB - which stores blocks of size 8 bytes to 1024 bytes, `large` for blocks sized between 1024 bytes and 512KiB, and `huge`, which stores any blocks larger than 512KiB. Segments store pages of the same kind, and are 4MB aligned. 64 small pages are stored in a segment, whereas `Large` or `huge` pages take up an entire segment - or multiple, in the case of `huge` allocations greater than segment size.

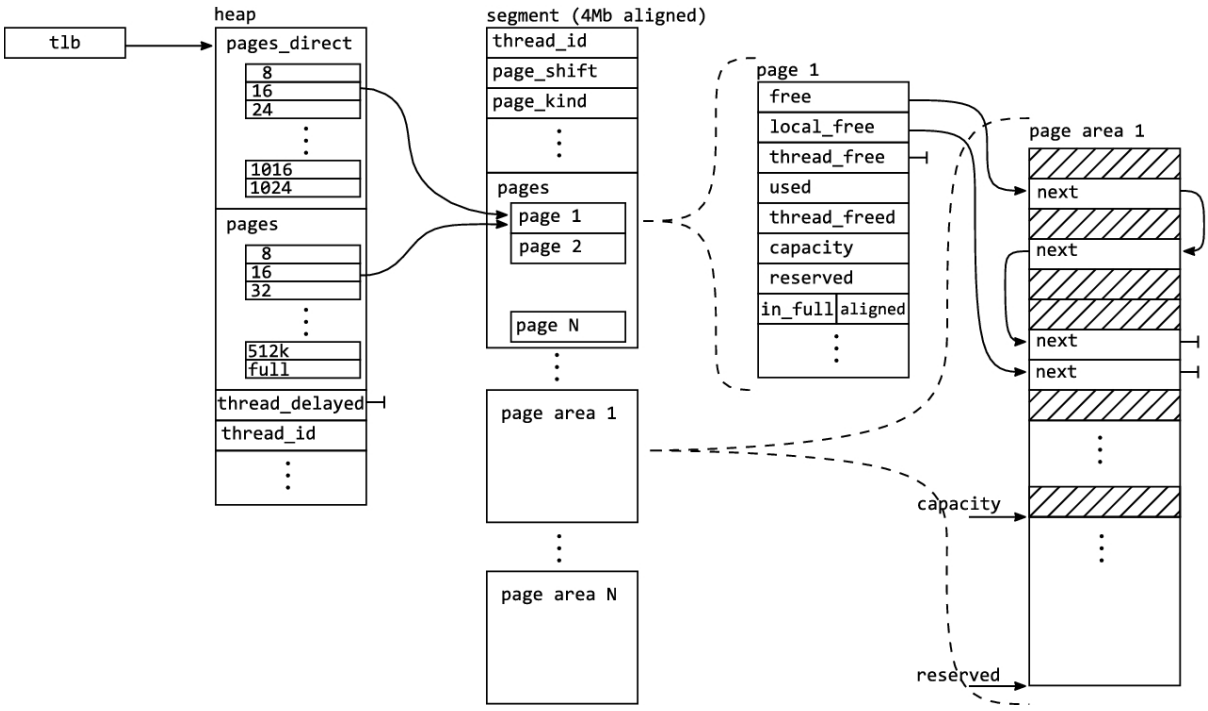

Figure 1: MiMalloc Heap Layout, taken from [3]

Malloc Strategy On a `malloc` call, CPU ID is obtained, which is then used to obtain a thread-local heap. From this point, there are two ways to access pages. Preferentially, `pages_direct` provides a direct pointer to a `small` page (for block size $\leq$ 1024, differentiated by increments of 8). In a more general case, `pages` is the head of a linked list of pages for a given block size that is a power of 2, starting at 8 bytes, and ending at 512KiB. Huge pages are accessed directly since they span an entire segment.

Free, and Reuse of Data Structures When a block of data is freed, but the page itself contains other in-use blocks, no further cleanup is done. Instead, as mentioned in the discussion on free list sharding the cleanup is done after a requisite number of calls to `malloc` - deterministically obtained via decrementing available free references in a page. If no blocks are in use at this point, the page itself is marked for reuse. Similarly, if no pages are in use for a given segment, the segment itself is marked for reuse. Segment allocation or freed status is represented by a bitmap `segment_bitmap`, which is protected by a lock to avoid race conditions.

Experimenting with Modifying MiMalloc Segment Size In MiMalloc, threads are assigned as "owners" of a contiguous 4MB chunk of memory, meaning that all calls to malloc from that thread will be attempted to be placed in the chunk. This has the benefit of improving locality and reducing false sharing. However, this creates a constant amount of overhead for every thread, since they must be assigned at least one 4MB segment regardless of the number or size of allocations. Additionally, segments are page-kind specific, meaning that if a thread allocates a single large (e.g. 4KB) and small (8 Byte) object, a small and large page would need to be created - so two segments would also be needed for the thread, resulting in 8 MB of overhead.

To optimize MiMalloc for the benchmarks used in testing "fragmentation" - which are hyper-sensitive to high constant memory overhead due to being a relative comparison to KHeap, we experimented with reduced segment size. As expected, this resulted in worse overall performance compared to 4MB-sized segments, but lower constant memory overhead. We determined that the tradeoff of static allocation in exchange for better performance was worthwhile so we kept the segment size at 4MB

2 Performance

All results found in the following figures were recorded using a host machine via remote-ssh into `b2240-01.teach.cs.toronto.edu`. This host machine had low usage, which was confirmed by running the `htop` command in terminal, and was running on 8 physical CPUs.

2.A False Sharing

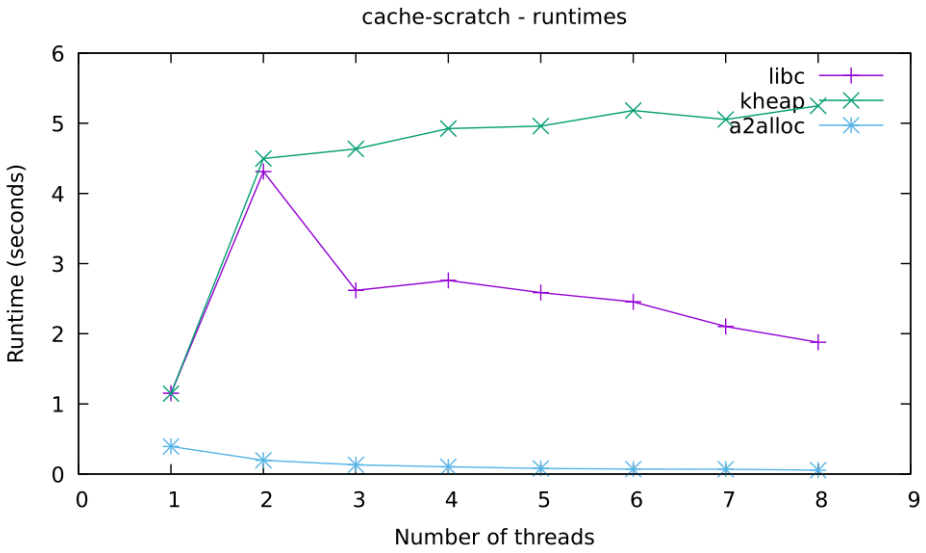

Figure 2: Cache Scratch Results - Runtimes

When a pointer is freed in our a2alloc implementation, the allocated block of memory returns to the owner thread's page by appending it to either the local-free or thread-free list, ensuring that only the owner thread can reallocate that chunk of memory while the page is in use. This implementation allows us to avoid passive false sharing because it does not break apart cache lines across threads. This can be seen in Fig. 2.A, where kheap and libc exhibit a sharp increase in runtime, indicating poor scalability due to passive false sharing, whereas a2alloc maintains a consistently low runtime across thread counts. This confirms our implementation avoids passive false sharing.

In our a2alloc implementation, each allocated page is "owned" by a segment, and each segment is "owned" by a thread. This implementation allows us to avoid active false sharing entirely, since no two threads can write to the same page, thus avoiding two threads writing to the same cache line. This can be seen in Fig. 2.A, where kheap exhibits a sharp increase in runtime, indicating poor scalability due to active false sharing. In contrast, libc and a2alloc maintain a consistently low runtime across thread counts. This confirms our implementation avoids active false sharing.

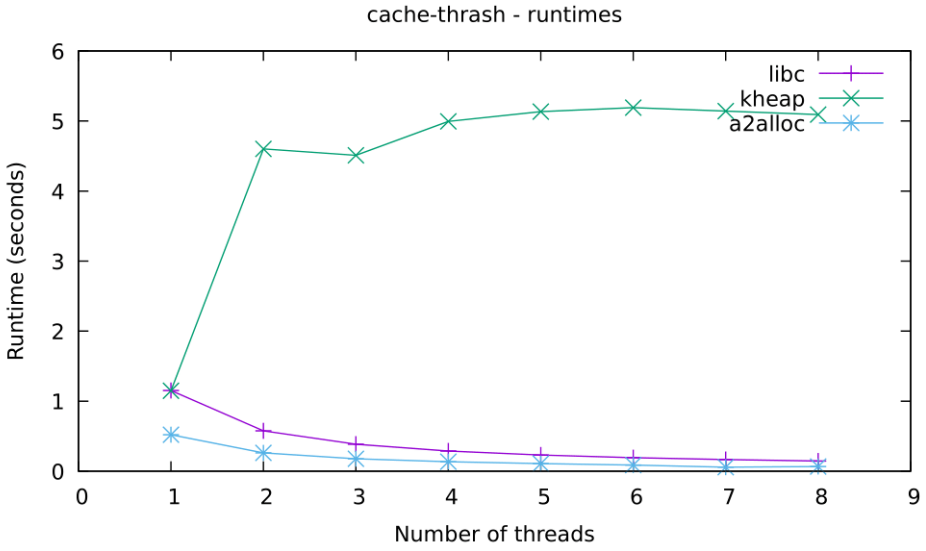

Figure 3: Cache Thrash Results - Runtimes

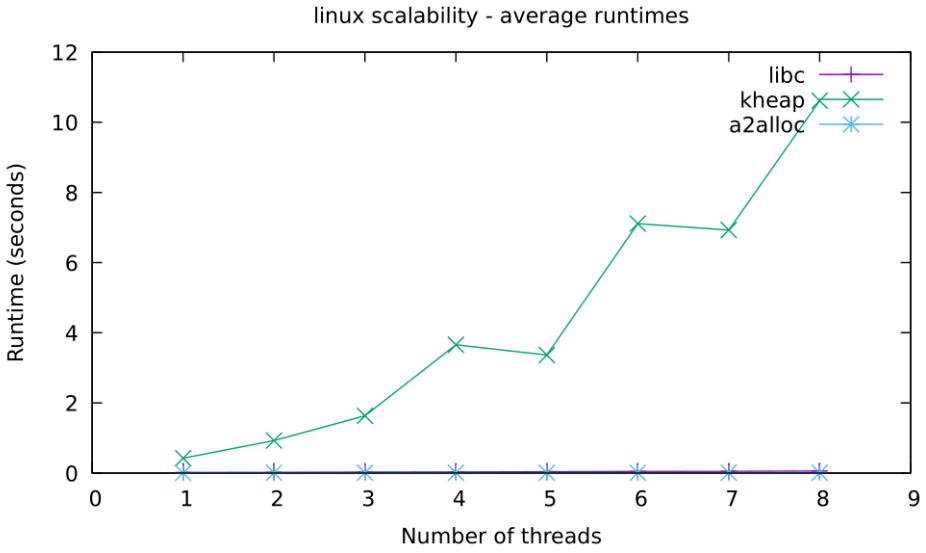

Figure 4: Linux Scalability Results - Runtimes

2.B Scalability

Figures 2.A, 2.A, 2.A, represent scalability tests. By increasing the amount of threads performing allocation and free operations, and possibly increasing the amount of work linearly with the amount of threads, these tests increase contention for shared resources. In our a2alloc implementation, we try to decrease contention by increasing the locality of our free lists, which reduces the number of concurrent accesses to the same blocks of data. Further, we decrease contention by deferring all atomic operations until they are necessary. The figures show that a2alloc achieves better runtime at all thread counts.

2.C Sequential Speed

In Figures 2.A, 2.A, 2.A, 2.A, 2.A, a2alloc outperforms both kheap and libc in terms of sequential speed. We see a lower runtime across all tests when we use one thread. This can be attributed to the way a2alloc defers expensive tasks until the free list of blocks is empty. This allows our implementation to defer expensive tasks, like reallocating blocks of free memory onto the page, until they are necessary.

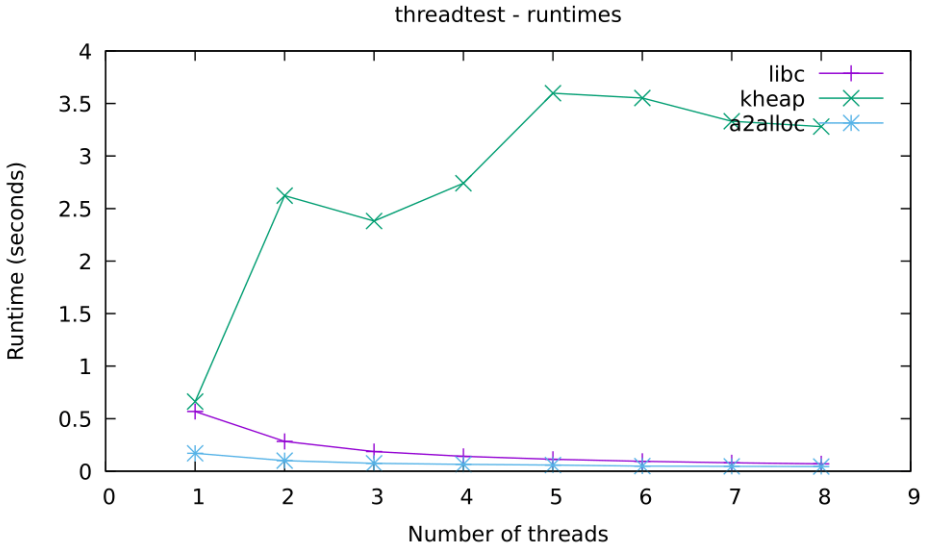

Figure 5: Thread Test Results - Runtimes

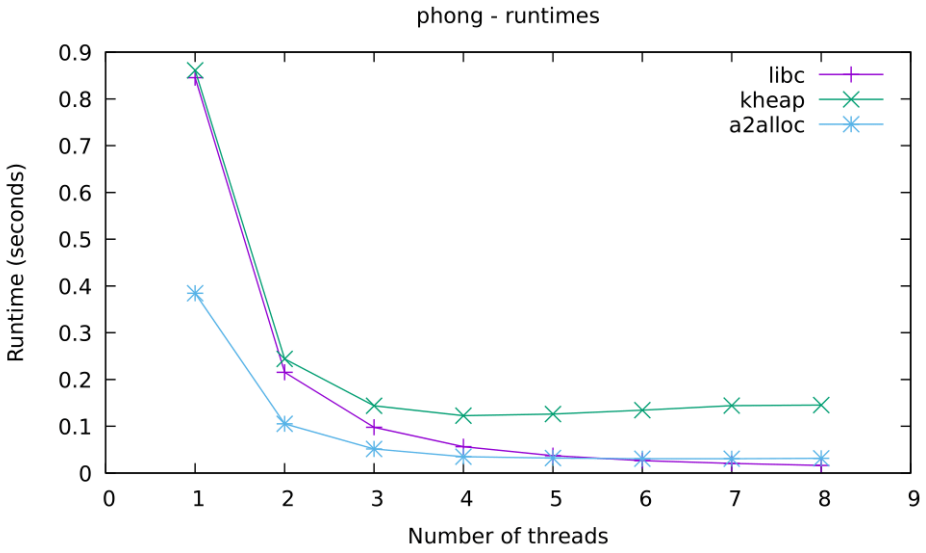

Figure 6: Phong Results - Runtimes

2.D Fragmentation and Memory Use

As seen in Fig. 2.D, a2alloc uses significantly more memory than kheap or libc. This increase in memory usage can be attributed to many design decisions made throughout the implementation. First, free list sharding increases the amount of memory overhead required to store free page information. Second, there is more heap metadata to store not only lists of pages, but also direct pointers to small pages. Also, whenever a2alloc requires new memory, we fetch 4MB at a time, resulting in potential overhead, as discussed in the design section. However, as mentioned, after testing smaller segment sizes, we found that decreasing segment size had a significant impact on performance and that the large segment size was a necessary tradeoff for the overall speed of our allocator.

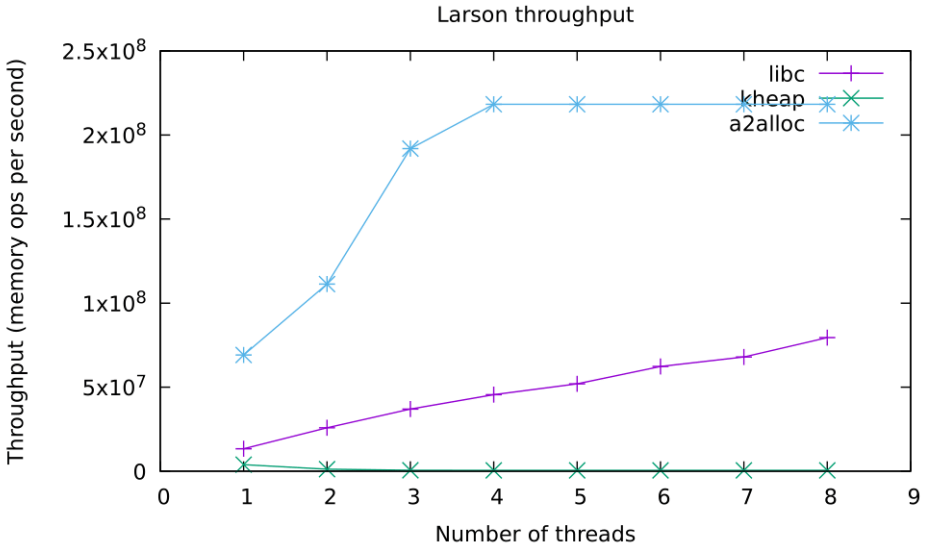

Figure 7: Larson Result - Throughput

Allocator	Cache Scratch	Cache Thrash	Larson	Linux Scalability	Phong	Thread Test
kheap	16383	12287	2813951	79859711	4263935	16383
libc	135168	135168	3244032	0	0	135168
a2alloc	35848191	35852287	35614719	171536383	103743487	36556799

Figure 8: Reported Memory Usage (Bytes) in Default-Configured Tests, 8 Threads

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[2] B. C. Kuszmaul, "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 International Symposium on Memory Management*, 2015, pp. 41–55.

[3] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 2019, pp. 244–265.